

UNITED STATES PATENT APPLICATION FOR:

**SYSTEM AND METHOD FOR DYNAMIC DATA BINDING
IN DISTRIBUTED APPLICATIONS**

Inventor:

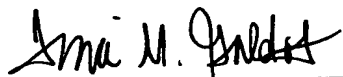
Edward K. O'Neil

**CERTIFICATE OF MAILING BY "EXPRESS MAIL"
UNDER 37 C.F.R. §1.10**

"Express Mail" mailing label number: EV385255188US

Date of Mailing: 2/17/04

I hereby certify that this correspondence is being deposited with the United States Postal Service, utilizing the "Express Mail Post Office to Addressee" service addressed to: **MAIL STOP: PATENT APPLICATION, Commissioner for Patents, P.O. Box 1450, Alexandria, VA 22313-1450**, and mailed on the above Date of Mailing with the above "Express Mail" mailing label number.



(Signature)

Name: Tina M. Galdos

Signature Date: 2/17/04

SYSTEM AND METHOD FOR DYNAMIC DATA BINDING IN DISTRIBUTED APPLICATIONS

Inventor:

Edward K. O'Neil

COPYRIGHT NOTICE

[0001] A portion of the disclosure of this patent document contains material which is subject to copyright protection. The copyright owner has no objection to the facsimile reproduction by anyone of the patent document or the patent disclosure, as it appears in the Patent and Trademark Office patent file or records, but otherwise reserves all copyright rights whatsoever.

CLAIM OF PRIORITY

[0002] This application claims priority from the following application, which is hereby incorporated by reference in its entirety:

[0003] U.S. Patent Application No. 60/450,516, SYSTEM AND METHOD FOR DYNAMIC DATA BINDING IN DISTRIBUTED APPLICATIONS, Inventor: Edward K. O'Neil, filed on February 26, 2003. (Attorney's Docket No. BEAS-1448US0)

FIELD OF THE DISCLOSURE

[0004] The present invention disclosure generally relates to dynamically binding web pages to server-side data.

BRIEF DESCRIPTION OF THE DRAWINGS

[0005] **Figure 1** is an illustration of exemplary logical system components in an embodiment.

[0006] **Figure 2** is an exemplary code segment to display information about the line items of a purchase order in an embodiment.

[0007] **Figure 3** is an exemplary code segment illustrating nested Repeaters in an embodiment.

[0008] **Figure 4** is an exemplary code segment illustrating the use of read/write JSP tags with a Repeater in an embodiment.

[0009] **Figure 5** is an exemplary code segment illustrating use of the Choice tag in an embodiment.

[0010] **Figure 6** is an exemplary code segment illustrating tags to render a paged, sortable, and filterable catalog data set in an embodiment.

[0011] **Figure 7** illustrates an exemplary structure of tags that the Grid can render in an embodiment.

DETAILED DESCRIPTION

[0012] Aspects of the invention are illustrated by way of example and not by way of limitation in the figures of the accompanying drawings in which like references indicate similar elements. It should be noted that references to “an” or “one” embodiment in this disclosure are not necessarily to the same embodiment, and such references mean at least one.

[0013] Data binding is the process of dynamically binding parts of a user interface (UI) to information. Data binding is important because in order to create useful business applications, a UI needs to present information to a user dynamically. Embodiments include a data binding infrastructure with three components: a set of data binding tags in a first programming language, and an expression language to specify data objects, and a third language to render the specified data objects.

[0014] **Figure 1** is an illustration of logical system components in an embodiment. Although this diagram depicts components as logically separate, such depiction is merely for illustrative purposes. It will be apparent to those skilled in the art that the components portrayed in this figure can be arbitrarily combined or divided into separate software, firmware and/or hardware. Furthermore, it will also be apparent to those skilled in the art that such components, regardless of how they are combined or divided, can execute on the same computing device or can be distributed among different computing devices connected by one or more networks or other suitable communication means.

[0015] In certain of these embodiments, data binding tags can be custom JSP (JavaServer® Page) tags **102** that have the ability through attributes to reference and

display data. By way of a non-limiting example, a text box tag can bind and submit data that a user may edit in a web page. Objects 106 encapsulate information (e.g., a JavaBean®, Java® object or XML document that contains information about a customer such as first name, last name, and shipping address would be a business object). The expression language 104 allows a software developer to set attributes on a JSP tag with an expression that references some field, element or property from an object. The data referenced with the expression may be displayed to the user or may be used to make decisions about what to display to a user. Information can be displayed using HTML (Hypertext Markup Language) 100 or other suitable means for rendering a UI.

[0016] In certain of these embodiments, the expression language – XScript – can be an extension to Javascript that adds a native understanding of XML (Extensible Markup Language) processing. XScript can use a subset of this language in its data binding expressions to reference properties on objects. By way of a non-limiting example, the following JSP actions will display a text box that contains an editable field for a user's zip code:

```
<netui:form action="ChangeAddress">
  <netui:textBox dataSource="{actionForm.shippingAddress.zipCode}"/>
  <netui:button>Submit</netui:button>
</netui:form>
```

[0017] The *dataSource* attribute on the text box tag references a property in the JavaBean containing the shipping address on the current action form; for information on action form objects, please see the page flow specification. In this case, the content of the *dataSource* attribute is an XScript expression. In one embodiment, an XScript expression is denoted by starting and ending the expression with the "{" and "}" characters respectively. In order to include these characters within an expression, they should be offset with a "\" character. An expression also has different parts, such as its context. In this case, the context of the expression is *actionForm*. The context defines the object in which the rest of the expression should be evaluated. In one embodiment, there are many different contexts in which to evaluate an expression including:

CONTEXT NAME	OBJECT THE CONTEXT REFERENCES
ActionForm	the action form that is associated with the current form tag.
PageFlow	the current pageFlow.
GlobalApp	the global app object for the webapp.
Request	the request's attribute map.
Session	the session's attribute map.
Application	the servlet context's attribute map.
url	the query string for the current request.
PageContext	a JSP page's attribute map.
Container	the complex data binding tag that is performing iteration over a data set

Table 1: Exemplary Contexts in an Embodiment

[0018] The *container* binding context is described below during the discussion of complex data binding tags. Not all binding contexts may be available at every stage of request processing. By way of a non-limiting example, the *pageContext* binding context cannot be referenced during the processing of a Page Flow action. Also, while some of these binding contexts are both readable and writable, others are simply readable. Specifically, the *actionForm*, *pageFlow*, and *globalApp* contexts can be written upon a request. The request processing lifecycle is described below.

[0019] On a *dataSource* attribute, an XScript expression can be what is known as an *l-value*. In one embodiment, an l-value is a programming language expression for something that sits on the left side of an assignment statement. Essentially, it is the part of an assignment that receives the assigned value. An l-value can also be used to reference a property on an object. XScript expressions can reference two parts of a Java object, the object's public fields and its public JavaBean properties. Public fields can be referenced as they are named in the class definition with no changes to the capitalization of the name. The restrictions on referencing public properties are a bit more specific. A public property is defined as a Java method with a *public* method signature, a non-void return value, zero parameters, and a method name that starts with *get*. An XScript

expression can reference any public method that meets these criteria. The JavaBean property name of a "getter" method can be used in the expression itself. By way of a non-limiting example, a property defined on an action form class with the signature:

```
public String getShipToZipCode();
```

is referenced in an expression as:

```
{actionForm.shipToZipCode}
```

[0020] The simple naming conventions of a JavaBean property are these:

- If the first two letters are upper case, remove "get" from the method name and leave the capitalization alone.
- Otherwise, remove "get" and lower case the first letter.

[0021] The result is the JavaBean property name. A property is read-only unless it has a corresponding JavaBean setter. A JavaBean setter is the inverse of the getter in that the method name can start with *set*, returns a void type, and has a single argument of the same type as the getter. If such a method exists on a class with the corresponding getter, the property is read / write.

[0022] Read / write properties are important in because such a property can be updated by the user from a web page. Generally, only simple types should be updated by the user, though those simple types may be properties of complex objects that may be properties of complex types and so on. Some simple types that the user can update would be String, integer, double, float, and boolean. In evaluating an expression, XScript can resolve properties on JavaBean objects (objects that expose properties as described above) and on Java array, List, and Map types. References to elements in an array or List are made through typical array indexing syntax:

```
{actionForm.purchaseOrder.lineItem[4].unitPrice}
```

[0023] This expression would reference the unit price of the fifth line item in a customer's purchase order. In order to reference items contained in a Java Map, the syntax is somewhat different:

```
{actionForm.databaseColumnValues["productname"]}
```

[0024] In this case, the field values for a database record are exposed for data binding through the property *databaseColumnValues* on an action form. This property returns a Map, and the *productname* property is extracted from the map.

[0025] In addition to supporting binding to JavaBean objects, XScript has the unique ability to bind directly to an XML document. XScript expressions can be written to traverse an XML document's elements and attributes in order to locate sets of or specific matching element(s). Thus, a web page can bind directly to XML with the same syntax that would be used to bind directly to Java. Note, XMLBeans are JavaBean representations of an XML document and provide another way to bind to an XML document in a strongly typed manner.

[0026] Business objects are any object in which a developer might store data that is important to the purpose of a web application. Basically, a business object contains data that is presented to and/or collected from the user; this data may originate from a database, XML document, LDAP server, SAP system, or elsewhere. XScript expressions are used to provide access to this object and its properties from the web pages that present data to the user. Virtually any business object that meets the property criteria described above is data-bindable, and developers are free to write arbitrary objects that can be data bound.

[0027] JSP tags can be used to present data from a business object to a user. Bridging the gap between the UI component and the business object, the XScript expression language can be used within attribute values on JSP tags to reference properties on a business object. A JSP tag can specifically render and provide the ability to update data from a business object. Attributes on a JSP tag that can contain an expression are referred to as *data-bindable*. In some cases, an attribute that is data-bindable can contain an XScript expression though in others data-bindable attributes may also contain literal text.

[0028] In one embodiment, tags can fall into two categories -- HTML and complex data binding. The HTML JSP tags can further be divided into two categories, read-only and read-write. The read-only JSP tags are those that simply read a value from some data source and render it, perhaps with additional formatting, in a web page. Examples of such tags are the *label* and the *content* tags where the former can be stylized and the latter simply renders unformatted content. The other half of the HTML JSP tags are those that correspond to elements in an HTML form and the form tag itself. By way of a non-limiting example, the text box, text area, select box, check box, and radio button tags are all examples of read-write JSP tags that render editable elements inside of a form tag. In relation to XScript and a business object, each of these tags can be used to update a single property on a business object. Some of these tags, such as the select box, can read from multiple properties on one or many JavaBean(s). In the case of the select box, it may read its list of options from one property, may read its default selected value from another, and may read / write its currently selected items into yet another property.

[0029] In the case of the read-only tags, these tags generally have a *value* attribute that may contain an XScript expression, but this attribute can sometimes also be literal text. By way of a non-limiting example, consider the following two label tags:

```
<netui:label value="{actionForm.firstName}"/>
<netui:label value="Jane"/>
```

[0030] In the first case, the label renders the value that is referenced by the expression *{actionForm.firstName}*, but in the latter case the text "Jane" is simply rendered. This is an example of a data-bindable attribute that can accept both literal text and an expression.

[0031] Read / write tags have an attribute called *dataSource* that references the "current" value that the tag should display to the user. The notion of a "current" value can be different depending on the tag. By way of a non-limiting example, on a text box tag the current value is the text to display. On a select box, the "current" value is the set of options that a user has selected, while the list of available options that are displayed is likely databound to a separate property on a business object. Regardless how a tag interprets the *dataSource* attribute, this attribute is usually required and should be an

XScript expression. For Struts compatibility, the *dataSource* attribute of the read / write tags may take a String that references a property on the current action form.

[0032] In one embodiment, if a JSP tag contains an expression, the expression is set as the value of the attribute rather than evaluating the expression first and setting the result as the value of the attribute. During the tag's lifecycle, expressions are evaluated as necessary and the result can be used internally to affect the rendering of the tag. In the case of read / write tags, the expression is also written to the web page in the *name* attribute of any HTML input tags. When the containing form is submitted to the server, the *name* attribute contains the XScript expression which is enough information to reference the property to update with the value that was displayed in the form.

[0033] The second set of tags are complex data binding tags which are used to render entire data sets to a web page. While the HTML tags generally render single valued objects to the page, e.g., a String text in a text box or a boolean in a check box, complex data binding tags are used to render entire arrays, lists, and maps where each item in the data set is rendered once. Often, the HTML tags described above are nested inside of the complex data binding tags so that meaningful UI is rendered for every item in a data set. Generally, the complex data binding tags can be thought of as tags that create rendering boundaries around the HTML JSP tags. A rendering boundary is established by the opening and closing of a JSP tag; the body contained inside is rendered at a specific time as defined by the container. Thus, the tag defines both a boundary for content and the rules for when that content is rendered. Complex data binding tags are also usually state-machine tags that render their body content several times before completing. While the HTML tags usually render their bodies exactly once, the complex tags render their bodies at least once for every item in a data set and sometimes many more times than that, and each pass through the body of such a tag may be in a different rendering state than the last. Rendering the body of a tag in a stateful way any contained tags that are aware of such stateful rendering to behave differently depending upon the current rendering state. Complex data binding tags are also usually tag sets with a top-level tag that contains other, well-known tags that work together to render a data set. The two tag sets of this type are the Repeater tag set and the Grid tag set.

[0034] In one embodiment, the Repeater tag set can be used to render arbitrary repeating mark-up as the tags themselves do not render mark-up internally. Rather, the

tags render mark-up that is provided by the user. The tag set consists of one top-level tag, the Repeater. Several well-known child tags can be contained within the Repeater tag that are used to further parameterize the Repeater and to offset additional rendering boundaries such as the header, item, and footer. The repeater tag can be used in one of two ways, unstructured or structured. In unstructured rendering mode, other tags that are part of the Repeater's tag set are not contained within the body, and the body of the tag is rendered once for each item in the data set. By way of a non-limiting example, the following would render each item in a String array on a separate line:

```
<netui-data binding:repeater dataSource="{actionForm.stringArray}">
  <netui:label value="{container.item}"/><br/>
</netui-data binding:repeater>
```

[0035] For each item in the String array referenced by the expression in the *dataSource* attribute, the item is rendered on its own line using a Label tag. In order to provide expression based access to the Repeater's current item, a binding context called *container* can be used within the body of the repeater. This binding context provides contained tags access to the nearest complex data binding tag, which exposes several properties through this binding context:

PROPERTY NAME	DESCRIPTION
Item	The current item from the data set.
Index	The current integer index for the current item in the data set.
Container	The nearest containing complex data binding tag.
Metadata	An optional attribute that may expose metadata about the data set.
DataSource	A reference to the data set that is being rendered; this is usually an XScript expression.

Table 2: Exemplary Container Properties in an Embodiment

[0036] These properties can be accessed from the *container* data binding context by using the XScript syntax described above. The label tag in the example above references the current item in the data set with the expression *container.item*. If the current item in the data set is a JavaBean, the properties of the JavaBean can be accessed by appending them to the end of this expression. Consider a Repeater that is displaying the line items in a purchase order:

```
<netui-data binding:repeater
    dataSource="{actionForm.purchaseOrder.lineItems}">
<netui:label value="{container.item.unitPrice}"/> <netui:label
    value="{container.item.supplier.supplierId}"/><br/>
</netui-data binding:repeater>
```

[0037] The unit price and supplier's unique identifier are displayed together on the same line. The Repeater's unstructured mode for rendering its body is convenient, but often when displaying a data set, each item in the data set needs to be rendered with greater structure.

[0038] In one embodiment, the Repeater's structured rendering mode can use three tags from its tag set -- the Header, Item, and Footer tags. These tags use the state-machine lifecycle of the top-level Repeater tag to render their bodies at a defined stage in the Repeater tag's structured rendering lifecycle. The header and footer tags are rendered exactly once at the start and end of the Repeater's lifecycle while the item tag is rendered once for each item in the data set. These tags used together provide the ability to create structured mark-up such as tables and lists in a web page.

[0039] **Figure 2** is an exemplary code segment to display information about the line items of a purchase order in an embodiment. In this example, a table is rendered that contains a header with titles, a row for each item that displays the product name, unit price, and customer's selected quantity, and a footer that closes the table. Here, the header and footer tags are not used for data binding, but in more complex examples, they might display metadata information in the header tag or might sum the price of the cart in the footer tag.

[0040] The Repeater tag can also be contained inside of itself. By way of a non-limiting example, two repeaters might be used to render each of the purchase orders that

were created during a business day. The outer Repeater would render the purchase order itself while the inner Repeater would render each of the line items in the purchase order.

[0041] **Figure 3** is an exemplary code segment illustrating nested Repeaters in an embodiment. In this example, the code from **Figure 2** is wrapped by a Repeater to display each purchase order. The result will be a table where every other row is a table of the line items in the purchase order. Note, however, the difference in the binding expression for the display of the individual line items. In the first example, the expression finds the line items in the *actionForm* context while in the second example, the line item array is found on the current purchase order using the Repeater's *container.item* syntax.

[0042] In addition to displaying read-only data as in these examples, the Repeater tag set can also contain read / write JSP tags including form elements such as text box, select, and check box. When using these tags in the body of a repeater that is contained within a Form tag, the result is a web page that allows editing the items in a data set.

[0043] **Figure 4** is an exemplary code segment illustrating the use of read/write JSP tags with a Repeater in an embodiment. This code allows a customer to examine their cart and edit the quantity of each item therein before placing an order. The quantity for each item is displayed in a text box that lets the user edit the value.

[0044] In one embodiment, when a form is submitted, each text box in the form submits its name and a possibly updated value to the Page Flow runtime. Because the text box is an editable field and its expression uses the *container* data binding context, its name is rewritten so that an absolute path to a JavaBean property is rendered into the web page for the text box's name attribute. By way of a non-limiting example, the text box for the third item above would render the expression:

```
{actionForm.purchaseOrder.lineItems[3].quantity}
```

[0045] The array index into the line item array is different for each row, and each row in the table contains a reference to a unique property in the action form object. In one embodiment, name rewriting works by examining the expression and qualifying the *container* contexts using the *dataSource* attribute on the nearest Repeater. In the case of nesting, if the *dataSource* of the nearest Repeater also references a *container* binding context, rewriting occurs at the next level of Repeaters.

[0046] In one embodiment, the Repeater provides more advanced features for the customization of how the items in the data set are rendered -- padding and choice. Padding can be used in the repeater to create a regular display of an irregular data set as the Repeater's rendering of its Item tag can be truncated or padded using the Pad tag. The Pad tag provides attributes such *maxRepeat*, *minRepeat*, and *padText* control rendering. MaxRepeat can be used to control the maximum number of that the Item should render while minRepeat controls the minimum number of times that the Item should be rendered. The padText can be used if the minRepeat value is not met with the size of the data set; then, the padText is rendered to bring the number of rendered items up to the value of minRepeat. The Pad tag is a peer to the Header, Item, and Footer tags and is useful when several repeaters need to appear to be the same size even if the sizes of their data sets are different.

[0047] The choice feature can be used to select the UI to render for each item in a data set; this consists of two parts -- a choice method and the choice options. The choice method can be used to decide which choice to render for the current item, and the choice options are named options that define the options for rendering the current item. The tags used to make the choice and define the choices are the ChoiceMethod and Choice tags, respectively. Both can be contained by the Item tag.

[0048] In one embodiment, the ChoiceMethod tag can appear once and the Choice tag can appear any number of times where each Choice tag defines a different UI to render for each item. The ChoiceMethod tag invokes a reflective call on an object, specified by an XScript expression, and can pass any data to the call including data from the current item in the Repeater. The result of invoking a ChoiceMethod tag is a String that is matched against the *value* attribute of each Choice tag. If a match is found, the body of the matching Choice tag is rendered for the current item; if no match is found but a Choice is marked as default, the default Choice is rendered.

[0049] **Figure 5** is an exemplary code segment illustrating use of the Choice tag in an embodiment. This example can be used to render a different UI based on the shipping status of each item in a purchase order. The ChoiceMethod tag invokes a method on the current Page Flow to decide which Choice to render given the value of the *shipState* property of the Repeater's current item. If the return value is *inTransit*, *arrived*,

or *notShipped* the associated Choice tags are rendered. Otherwise, the last, default Choice is rendered.

[0050] In one embodiment, the Grid tag set is similar to the Repeater tag set in that it renders a data set; however, the Grid tag set expects regular data and takes a column-oriented approach to rendering the data set. Grids can also provide advanced functionality including the ability to sort, filter, and page through a data set. While the Repeater exposes the current data item directly to its child tags, the Grid requires that a particular interface be implemented overtop each type of data set that may be rendered. In order to expose features such as filtering, the Grid requires metadata about the data set that is not exposed through the properties on the data set object.

[0051] In one embodiment, the *dataSource* attribute on the Grid tag requires that referenced data set be supported by an implementation of a DataContext interface. The DataContext interface exposes the data in the items of the result set, an iteration mechanism for iterating over this data, and metadata about the underlying data set including type and unique identifier information. In one embodiment, the DataContext interface can support rendering a *javax.sql.RowSet* with the Grid tag set. The RowSet interface is a disconnected view of database data that is created from the more common *java.sql.ResultSet*, which is usually returned when executing a query on a database through JDBC (Java Database Connectivity). The RowSet extracts the data from the Result and stores it internally; the DataContext interface is aware of how to extract data and metadata from the RowSet interface.

[0052] The Grid takes a column-centric approach to rendering data. In addition, the Grid tag renders its own table, rows, and cells that wrap records contained within the data set. While the Repeater can render each row in a data set with a different UI, the Grid renders each row in the data set using the same UI. The UI for the entire grid, including the header, rows, and footer can be defined with a set of GridColumn JSP tags. GridColumn tags are used to output the cells contained within a single, vertical column in the HTML table that the Grid renders. Each GridColumn tag can render a column specific header that might be plain text or might include links to sort or filter a particular column. Generally, a GridColumn tag references the data that it will render by name. The DataContext interface exposes methods that allow the Grid to extract data from the current item in the DataContext by name. For the RowSet implementation of the

DataContext, this name can be the column name from the database table that contained the RowSet's data.

[0053] In one embodiment, a BasicColumn can be used to simply render data from the data set and is the most commonly used column type. An AnchorColumn optionally references a specific cell in the current data item and can be used to render a link that to another page or to an action in a Page Flow. If the name of a data item is not specified on the AnchorColumn, its title is rendered in every cell with the appropriate link. An ExpressionColumn allows a developer to provide an expression that will be evaluated to customize the appearance of a column. In the Basic and Anchor Column tags, the data that is present in the data set is rendered directly and no visual customization, except through the use of Formatter tags, is allowed as the tags never expose the data to the user as in the Repeater.

[0054] In one embodiment, the Basic and Anchor Column tags extend from the SortFilterColumn, which can render links in the header of the Grid that, if clicked, will sort the column or allow the user to specify a filter for the column. The process of rendering and maintaining sorts and filters can be done through query parameters on the URL; a separate class called the SortFilterService can be used to provide an API over the sorts, filters, and paging characteristics of the grid. By way of a non-limiting example, in order to render a link to change the page of the grid, all of the current sorts and filters can be maintained on this link. The SortFilterService exposes an API that allows a column implementation to ask for a link that will maintain the sorts and filters but change to a different page. Similarly, the SortFilterService provides methods to change the current sort on the URL.

[0055] In one embodiment, the link rendered for a filter is Javascript that pops-up a filter window that allows the user of a web page to constrain the values on a column. The Grid supports two levels of filters and a filter on each column that is filterable. The process of actually sorting or filtering the data is done on a server, however. On the server, a SortFilterService can be used to create a DatabaseFilter object that is aware of the sorts and filters in the URL and can generate the appropriate SQL for the ORDER BY and WHERE clauses of the SQL (Structured Query Language) statement. The DatabaseFilter object can be used in conjunction with a database control to parameterize a query to the database and return a RowSet that reflects the constraints specified in the

URL. Any other capabilities of filtering, including clearing the filters of a column and clearing all the filters of the grid, are performed by setting the URL of a Grid's window to a value that simply does not contain these parameters in the URL.

[0056] In one embodiment, the Grid's tag set includes a Columns tag, Pager tag, and a GridStyle tag. The Columns tag can be used to contain all of the columns that will be rendered for the grid and acts as a rendering boundary for these tags. In addition, the global capabilities for sorts and filters can be set on the Columns tag for the contained grid column tags that can be sorted and filtered. The GridStyle tag can be used to set the table, row, alternating row, header, and footer style classes that are rendered in the Grid. If an alternating row style is specified, this style will be applied to all of the odd numbered rows in a Grid page. The Pager tag can be used to parameterize the Pager that will be rendered by the grid; the pager can be used to display a reasonable number of rows on a particular page and allow the user of a web page to navigate through pages by using the common "Previous" and "Next" idiom. The Pager tag has boolean properties that allow a Grid to be rendered before and / or after the HTML table that the Grid renders to display the data. **Figure 6** is an exemplary code segment illustrating tags to render a paged, sortable, and filterable catalog data set in an embodiment.

[0057] The Grid tag set renders similarly to the Repeater because it uses a state machine and cooperation with the Grid tag set's tags to perform particular actions during specific render states. The Grid lifecycle is somewhat more complicated because the single, contained iteration boundary, the Columns tag, renders during multiple states and performs a different action for each one. **Figure 7** illustrates the structure of the tags that the Grid can render in an embodiment.

[0058] In one embodiment, the Grid passes through the *HEAD_PAGER*, *HEADER*, *ROWS*, *FOOTER*, and *FOOT_PAGER* in the process of rendering a Grid into the structure above. The Columns tag renders its body once in the HEADER state, once for each item in the page in the ROWS state, and once for each item in the FOOTER state, and each of the grid column tags contained within the Columns tags renders once for each one of these states. The GridColumns tag defines a rendering method for each of the HEADER, ROWS, and FOOTER states and invokes the appropriate method for the current state:


```
protected abstract String renderHeader();
protected abstract String renderCell();
protected abstract String renderFooter();
```

[0059] A GridColumn class can provide a plug-in point for developers who need to write their own column implementations. All of the data in each column is rendered through one of these methods depending on the lifecycle state of the column.

[0060] The tag set provides a set of abstract base tags that can be extended to provide a plug-in point for developers that need to write tags that can participate in the Page Flow runtime and in data binding. In one embodiment, read / write tags render an HTML `<input type="" ... >` tag to the web page, and these tags contain a *name* attribute that is sent in the request when the containing form is submitted back to the server. The *name* attribute of the tag is present in the request object's parameter map at the server; the value associated with each name key is the value that was submitted by each of the input tags. If the value of a form element was edited, this value will contain the edit. In addition to using the XScript expression in the *dataSource* attribute for reading data, this expression is also used to update the property referenced by the expression. The value of the data source attribute is written into the *name* attribute of the HTML input tag. Then, when the form is submitted, the expressions are submitted with the current value of the associated form element.

[0061] By way of a non-limiting example, if a text box is rendered with the expression `"{actionForm.firstName}"` and the user types "Jane", the value submitted to the server for this text box is the pair `"{actionForm.firstName}"` and "Jane". If the form containing the JSP tags is submitted to a Page Flow action, the Page Flow runtime starts processing the request. At an early stage in this processing lifecycle, the values in the request are used to update the underlying business objects. This can be performed by iterating over each of the "key" values that were submitted in the request. In this example, the XScript expression would be the key for the value "Jane". If the key is an XScript expression, the property referenced by the expression is updated to equal the new value. Because a request represents all submitted values as Strings, XScript can infer the type of the property that is being set and converts the value of each XScript expression key to the target type and then sets this value on the property referenced by the expression. Once

all of the XScript expressions have been updated, all Struts properties are updated using the Struts request processing mechanism.

[0062] In one embodiment, data binding contexts such as *actionForm* and *container* are defined by the ContextResolver interface and with a context name. When an expression is parsed, its context can be used to find an associated ContextResolver. If a matching ContextResolver is found the ContextResolver is passed a Map of available objects and is expected to create an XScript Scriptable object that represents the given context. The next identifier in the context is evaluated against this Scriptable object and expression evaluation continues until the end of the expression.

[0063] In one embodiment, ContextResolvers can have a unique name in the global namespace of available contexts. A ContextResolver implementation can be marked as request read-only or request read / write; by default, all context resolvers are request read-only though some contexts such as the *actionForm* and the *pageFlow* are read / write. A request read-only context is a context that can not be updated when data submitted from a web page form is being processed. Allowing all contexts to be updateable would create a security vulnerability as read / write contexts could be referenced with large amounts of data that could be used to perform a denial of service attack on a server. In the Page Flow runtime, however, most expression contexts such as the *request* and the *session* are updatable by invoking the expression evaluation engine directly. The contexts that are available in the server can be specified through a context.properties file that is parsed when the first expression is evaluated. This file establishes the context name to ContextResolver implementation mappings.

[0064] In one embodiment, when a form is submitted from a web page, the (key, value) pairs that are submitted to the server from the editable form components are represented entirely as String data. If the key of such a pair is an XScript expression, the expression may reference a property on a business object of any type. Thus, type conversion can be used to change the String value into the type that the property's setter expects. Type conversion can handle all of the basic primitive, primitive wrapper, String, and some database types including BigDecimal and Date. Additional types or overrides of the default converter implementations can be plugged-in by a web application developer by creating or editing a Java property file in the webapp's WEB-INF directory. When the web application is deployed by the server, this property file is parsed and the

specified type converters are loaded into the TypeUtils class. XScript performs type inferencing on the object referenced by the submitted expression, the type of the referenced property is discovered, and the key's associated value and this type are passed to the type converter. The return value is an Object that can be reflectively applied to the setter for the property.

[0065] In one embodiment, the DataAccessProvider interface is implemented by the Repeater tag and can be used to provide tags in the Repeater's body access to properties on the *container* binding context. Repeater nesting and referencing the properties of ancestral Repeater's is possible because of the Repeater's implementation of this interface; the context that evaluates the *container* expressions is aware of the DataAccessProvider interface and can navigate up the DataAccessProvider / Repeater hierarchy in order to resolve properties on parent Repeater tags.

[0066] Rewriting the *dataSource* attributes of read / write tags contained within a Repeater is can be a non-trivial process. Any references to the *container* binding context can be resolved into fully qualified XScript expressions that reference a specific property. After rewriting such an expression, the result should be usable outside of a Repeater to bind directly to the property referenced from the *container* binding expression. Expression rewriting is complicated by the fact that Repeater tags can be deeply nested and that tags inside of a repeater can refer to properties on the parent using the syntax *container.container* and the Repeater's parent's current data item is available with the binding expression *container.container.item*.

[0067] Name rewriting can start with the expression to rewrite, for example *container.item.firstName*. If the *dataSource* of the *container* is a real reference as *actionForm.customer*, then it is substituted to create the indexed expression *actionForm.customer[index].firstName* where the index is the current index into the data set. If instead the *container*'s reference contains a reference to another *container*, then the expression can be rewritten using the parent's parent's *dataSource* attribute, and so on. Additionally, if an expression starts with *container.container.item.firstName*, the rewriting can start with Repeater's parent's *dataSource* attribute.

[0068] In one embodiment, a Repeater's rendering lifecycle can include several stages: *HEADER*, *ITEM*, and *FOOTER*. As the Repeater renders its body, it walks these stages in order and renders in *HEADER* state once, *ITEM* state once for each item in the

data set, and FOOTER state once at the end of the Repeater lifecycle. Within ITEM state, the Item tag makes two passes through its body once in each of the *CHOOSE* and the *RENDER* states. In *CHOOSE* state, the ChoiceMethod tag executes and any Choice tags register their value and default attributes. At the end of the *CHOOSE* state, the Item tag can decide which Choice tag to render as it knows the values of all of the choice tags. In *RENDER* state, the Choice tags each ask the Item for permission to render, and if granted, a Choice tag will render its body which can databind to the Repeater's current item. If no ChoiceMethod or Choice tags are present, the Item renders its body on the first pass and the render state of the Item tag is ignored by the contained tags.

[0069] One embodiment may be implemented using a conventional general purpose or a specialized digital computer or microprocessor(s) programmed according to the teachings of the present disclosure, as will be apparent to those skilled in the computer art. Appropriate software coding can readily be prepared by skilled programmers based on the teachings of the present disclosure, as will be apparent to those skilled in the software art. The invention may also be implemented by the preparation of integrated circuits or by interconnecting an appropriate network of conventional component circuits, as will be readily apparent to those skilled in the art.

[0070] One embodiment includes a computer program product which is a storage medium (media) having instructions stored thereon/in which can be used to program a computer to perform any of the features presented herein. The storage medium can include, but is not limited to, any type of disk including floppy disks, optical discs, DVD, CD-ROMs, microdrive, and magneto-optical disks, ROMs, RAMs, EPROMs, EEPROMs, DRAMs, VRAMs, flash memory devices, magnetic or optical cards, nanosystems (including molecular memory ICs), or any type of media or device suitable for storing instructions and/or data.

[0071] Stored on any one of the computer readable medium (media), the present invention includes software for controlling both the hardware of the general purpose/specialized computer or microprocessor, and for enabling the computer or microprocessor to interact with a human user or other mechanism utilizing the results of the present invention. Such software may include, but is not limited to, device drivers, operating systems, execution environments/containers, and applications.

[0072] The foregoing description of the preferred embodiments of the present invention have been provided for the purposes of illustration and description. It is not intended to be exhaustive or to limit the invention to the precise forms disclosed. Many modifications and variations will be apparent to the practitioner skilled in the art. Embodiments were chosen and described in order to best explain the principles of the invention and its practical application, thereby enabling others skilled in the art to understand the invention, the various embodiments and with various modifications that are suited to the particular use contemplated. It is intended that the scope of the invention be defined by the following claims and their equivalents.